LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-CONF-219596

Dynamic Program Phase Detection in Distributed Shared-Memory Multiprocessors

E. Ipek, J. F. Martinez, B. R. de Supinski, S. A. McKee, M. Schulz

March 7, 2006

IPDPS Workhop on the NSF Next Generation Software Program
Rhodes, Greece
April 29, 2005 through April 29, 2005

Dynamic Program Phase Detection in Distributed Shared-Memory Multiprocessors

Engin İpek[1] José F. Martínez[1] Bronis R. de Supinski[2] Sally A. McKee[1] Martin Schulz[2]

[1] Computer Systems Laboratory
Cornell University
Ithaca, NY 14853 USA
{engin,martinez,sam}@csl.cornell.edu

[2] Center for Advanced Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA 94551 USA
{bronis,schulz}@llnl.gov

***Abstract*— We present a novel hardware mechanism for dynamic program phase detection in distributed shared-memory (DSM) multiprocessors. We show that successful hardware mechanisms for phase detection in uniprocessors do not necessarily work well in DSM systems, since they lack the ability to incorporate the parallel application's global execution information and memory access behavior based on data distribution. We then propose a hardware extension to a well-known uniprocessor mechanism that significantly improves phase detection in the context of DSM multiprocessors. The resulting mechanism is modest in size and complexity, and is transparent to the parallel application.**

I. Introduction

Analyzing the time-varying behavior of applications has been the subject of several studies [5], [7], demonstrating that relying on average whole-program statistics can lead to misconceptions about a program's actual behavior, and result in poor architecture optimization decisions. Yet in spite of such behavior changes over a program's entire execution, application behavior is typically repetitive, and can be classified into distinct *phases*—collections of dynamic execution regions, not necessarily consecutive, exhibiting similar behavior and thus requiring similar resources. In phase-adaptive systems, hardware *phase detectors* monitor runtime metrics at the granularity of fixed sampling intervals, and classify these intervals into program phases [4], [8] to guide hardware reconfiguration.

Hardware phase detection has been studied extensively for uniprocessors. To our knowledge, however, no published work yet discusses general solutions to hardware phase detection in parallel systems. In this paper, we address hardware phase detection in the context of distributed shared-memory (DSM) multiprocessors. Specifically, we:

- Illustrate how uniprocessor approaches that only consider instruction working sets are generally not portable to DSM multiprocessor environments, and the quality of their phase detection degrades quickly with the system size.
- Propose and evaluate a low-overhead architectural mechanism that captures data distribution, latency, and contention effects of a DSM multiprocessor setting. This results in markedly improved phase detection for a variety of parallel applications.

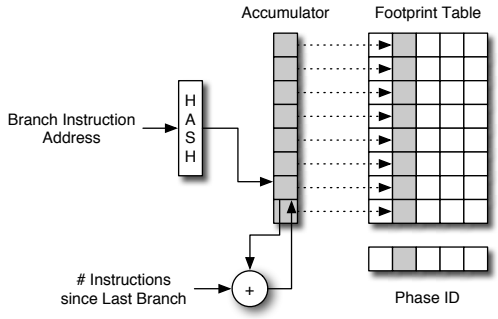

Fig. 1. Example of BBV phase detector.

- Introduce a new tool, the *CoV curve*, that helps quantify the quality of phase detection of a particular mechanism across multiple operating points.

II. Overview

In phase-adaptive systems, a phase detector collects program statistics at runtime and, at regular sampling intervals, determines whether the program incurred a phase change. This information is passed to a phase predictor, which infers the phase for the next sampling interval. Finally, a reconfiguration module tunes the system based on this prediction, by trying different hardware configurations at different intervals that belong to the same phase. Once tuning is complete, the best configuration is selected, and subsequently applied whenever that phase is predicted. This trial-and-error process may hurt performance, and thus it must be conducted efficiently.

Our baseline uniprocessor phase detector is Sherwood et al.'s BBV mechanism [8] (Fig. 1). Intervals are classified into phases based on execution frequencies of their basic blocks. This information is tracked using an array of hardware counters called *accumulator*, which is hashed by branch instruction addresses, and used as a vector to compute the Manhattan distance between it and every vector in a footprint table recording previously calculated accumulators. If the distance to the closest footprint vector is smaller than a certain *distance threshold*, the interval is classified into the same phase. Otherwise, the interval is regarded as an instance of a new phase, and the accumulator is transferred to a footprint vector.

A statistical metric for evaluating phase detectors is the *Coefficient of Variance* (CoV). For a given program phase, its CoV of CPI (or simply CoV) is the ratio of the standard deviation to the mean of the all the per-interval

CPI values in that phase. The *identifier CoV* is then defined as the average of all per-phase CoVs, weighted by how many intervals belong to each phase. A phase detector that classifies intervals into perfectly homogeneous phases yields a CoV of zero, and CoV increases as phases deviate from this ideal interval homogeneity.

CoV is naturally smaller with a higher number of phases, since fewer intervals belong to each phase—in the extreme case, every sampling interval would constitute a distinct phase (each requiring tuning), with CoV trivially zero. Conversely, all sampling intervals could be placed in the same phase, and thus tuning overhead would be minimal—but most likely futile, as the resulting CoV would be prohibitively large. To quantify this trade-off, in the next section we introduce the *CoV curve*, which plots CoV against a measure of tuning overhead (the fraction of intervals that are spent in tuning).

III. Phase Detection in DSM

We conduct detailed simulations of a DSM multiprocessor with up to 32 nodes. Table I shows several architectural parameters of the system we model. We use two applications from the SPLASH-2 suite (LU and FMM) [10] and two applications from the SPEC-OMP suite (Art and Equake) [9]. Table II lists the applications and input sets.

TABLE I
SUMMARY OF SIMULATED ARCHITECTURE.

Parameter	*Value*
Processor Frequency	2GHz
Functional Units	6 ALU, 4 FPU
Fetch/Issue/Commit	6/6/6
Register File	128 Int, 128 FP
Branch Predictor	2,048-entry gshare
L1	16kB, direct-mapped, 1 cycle
L2	2MB, 8-way, 32B, 12 cycles
Memory	SDRAM interleaved, 75ns, 2.6GB/s
Network	Hypercube, wormhole, 400MHz pipelined router, 16ns pin-to-pin

TABLE II
APPLICATIONS USED IN THE EXPERIMENTS.

Application	*Input Set*
LU	512×512 matrix, 16×16 block
FMM	65,536 particles
Art	Minnespec–Large
Equake	Minnespec–Large

A. Uniprocessor Scheme on DSM

We evaluate the efficacy of the BBV uniprocessor scheme in a DSM environment by adding a BBV phase detector to each processor in our simulation framework (Section III). Each detector consists of a 32-entry accumulator and a 32-vector footprint table. We use a LRU replacement algorithm. The interval size in each processor is 3M committed non-synchronization instructions, divided by the number of processors in each configuration, so that configurations with similar number of phases also have similar number of intervals per phase (and thus tuning overhead) even as the system size scales up (and thus each processor executes less code). We examine two hundred threshold values. We compute identifier CoV curves for each processor, and then average them together to obtain the overall system-wide CoV curve. Figure 2 shows the results.

As expected, when the BBV phase detector is applied to a DSM context, it classifies intervals poorly as the number of processors grows. For instance, with two processors, LU achieves CoV values under 10% with as few as seven phases.[1] At eight and 32 nodes, however, the CoV value for seven phases rises dramatically to about 40 and 70%, respectively. In fact, when running on eight and 32 processors, LU only achieves a 20% CoV with 25 phases—a two-fold degradation with nearly four times as many phases as the two-processor case explained above.

In summary, even as the BBV mechanism has been shown to successfully characterize the behavior of sequential applications by simply tracking the distribution of executed basic blocks [8], several factors limit its effectiveness in a DSM environment running parallel codes. First, the behavior of a thread may be affected by the other threads executing in the system by means of data sharing patterns, memory traffic, network congestion, etc. Second, data distribution (e.g., local vs. remote accesses) may affect the behavior of the code executing on a node, even when a processor executes precisely the same code without interaction with others. Unfortunately, the BBV alone cannot (and was never meant to) capture these multiprocessor-specific factors.

B. DSM Phase Detector

We propose a per-processor, on-chip *data distribution vector (DDV)* that extends the BBV to track data distribution and contention. Each DDV contains a *frequency* matrix F, a *distance* matrix D, and a *contention* vector C. Figure 3 shows an example of a BBV+DDV-based phase detector for one processor in a two-processor system.

Committed memory accesses are accounted for in the frequency matrix, according to the home of the memory block they access.[2] This is available to the on-chip memory controller, particularly since the access has already taken place at the time the memory operation commits (and thus any TLB miss or other address translation step has already been serviced).

This information is queried by all processors, regularly on a per-interval basis. Notice, however, that intervals are defined independently by each processor, based on the number of (locally) committed instructions. Hence, to convey information that is consistent with the requestor's interval boundaries, each processor keeps separate frequency counts of its accesses on behalf of each processor in the system. Such counts are zeroed every time the corresponding processor queries their content (see below).

Specifically, in an n-processor system, each frequency matrix has n n-entry frequency vectors F_i—one per processor, including itself—, for a total of $n \times n$ frequency counters. At each processor p, counter F_{ij} keeps track, on behalf of processor i, of the number of loads and stores

[1] Recall that fewer phases generally imply less tuning overhead.

[2] Factors that are likely to be tainted by device reconfiguration, such as whether the access hits or misses in the cache, are not considered.

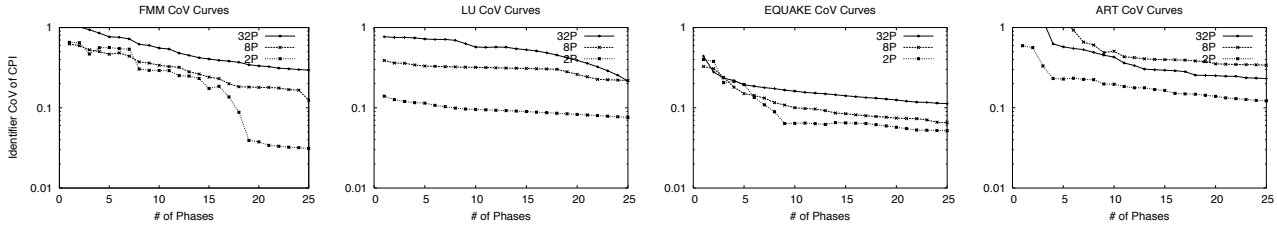

Fig. 2. Baseline BBV results. Note the logarithmic scale on the y axis.

committed by processor p that accessed data with home in node j *since processor* i *last started a new interval*.

Every time processor p commits a load or a store operation that accesses data with home in node j, it increments all F_{kj}, $1 \leq k \leq n$. When a processor i reaches the end of an interval, it requests all F_i vectors in the system. As other processors hand out their F_i vectors, they each reset their local copy, thus initiating a fresh count for the next interval on behalf of processor i. Meanwhile, processor i adds all n vectors (including its own) into its n-entry contention vector C. Then, it computes its *data distribution scalar value* (DDS) locally as follows:

$$\text{DDS} = \sum_{j=0}^{n-1} F_{ij} D_{ij} C_j$$

where D_{ij} is a measure of the distance from node i to node j (1 if $i = j$). (D is a matrix of pre-programmed constants.) Each term j of the summation captures, for the interval under consideration, (1) the frequency with which i accessed data with home j, (2) the distance between nodes i and j, and (3) the system-wide contention for data with home in j. The overall sum is a measure of i's "cost" of accessing data during that interval.

After i computes its DDS, it uses it in conjunction with its BBV accumulator to find its phase in the footprint table. Specifically, the processor uses one BBV *and* one DDS distance threshold. If one or more entries in the footprint table yield both an accumulator Manhattan distance and a DDS difference below their respective pre-set thresholds, the entry with the smallest Manhattan distance is taken. If no footprint table entry satisfies this condition, a new entry is allocated, possibly replacing an old one, and both accumulator and DDS are stored. Finally, i resets its own F_i vector (as well as its accumulator), and a new interval begins for i.

The entire process is handled by dedicated hardware in each node, and we envision the BBV+DDV structures stored in a small, dedicated on-chip memory module, which allows system scalability (e.g., by sizing DDV structures depending on n) and even multiprogramming (e.g., by replicating such structures, as we briefly address later). More details on the actual hardware implementation are beyond the scope of this paper.

The communication cost involved in the computation of DDS by processor i is $n-1$ exchanges with as many processors. Assuming 32 2GHz processors, IPC = 1, and

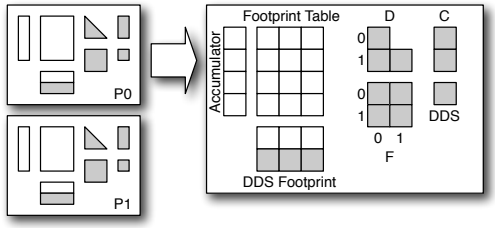

Fig. 3. Example of a BBV+DDV-based phase detector of a processor in a two-processor system. Shaded objects denote DDV hardware. In the figure, F, D, and C stand for Frequency matrix, Distance matrix, and Contention vector, respectively.

a "real-world" interval length of 100M instructions,[3] the overall sustained bandwidth requirement of this mechanism is about 160kB/s. If modern memory controllers can handle 1.5GB/s, then the overhead of this mechanism is under 0.15% of the peak bandwidth. Thus, we expect the mechanism to account for a negligible overhead in systems of that scale.

We briefly comment on a couple of other issues:

- Our phase tracking mechanism can capture parallel program behavior under both static and dynamic models of execution. For instance, in a centralized task queue implementation, variations in the distribution or contention of the data accessed across tasks by a processor would be captured by the DDV.
- In a multiprogrammed environment, the phase identification information can be incorporated into the thread's state on a context switch. Alternatively, phase information associated with threads can be cleared at the expense of more tuning.

IV. Evaluation

To evaluate our proposed phase detector, we plot CoV curves for eight and 32 nodes for the four applications under study. The sampling interval size at each processor is 3M committed non-synchronization instructions, divided by the number of nodes in the system in each case. For both BBV and BBV+DDV phase detectors, we use a per-node BBV accumulator vector of 32 entries, backed by a 32-vector footprint table. Figure 4 shows the results.

When distance thresholds are high enough that the entire program falls into a single phase, both detectors naturally achieve the same CoV result. As the distance threshold decreases, however, the number of phases increases, and differences between CoV curves emerge as a result of

[3]Due to the reduced input set sizes of our applications, we use 3M instruction intervals in our studies.

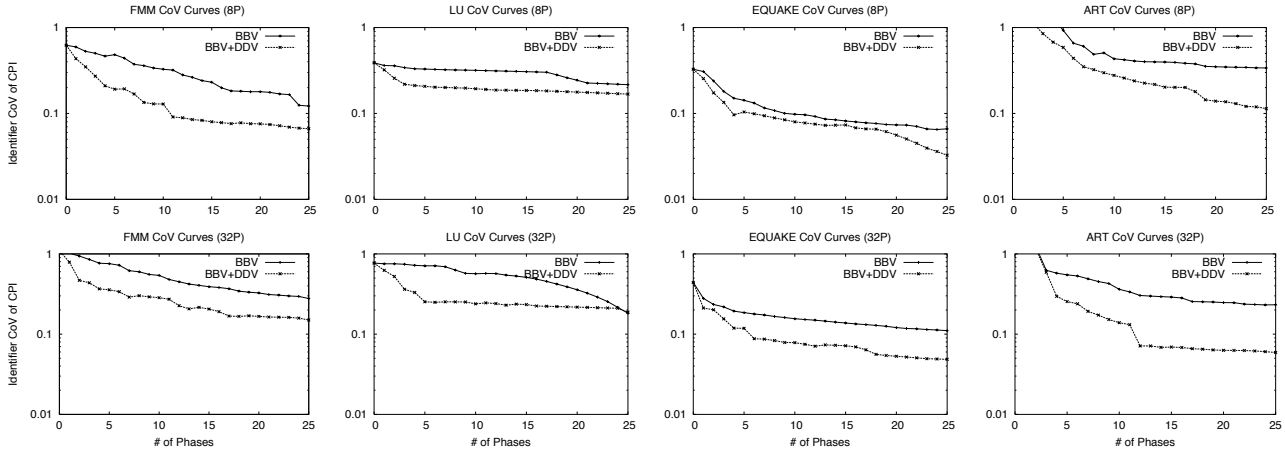

Fig. 4. BBV+DDV results. Notice the logarithmic scale on the *y* axis.

the different classification criteria. The BBV+DDV configuration improves the CoV values achieved by the BBV baseline across the board. Moreover, the benefits of using the DDV over the BBV baseline increase with the number of processors, as expected, since an increased node count implies *(a)* more and longer accesses to remote data, and *(b)* higher variability due to interactions among threads, which are captured by the DDV.

In FMM, for example, with 32 processors, BBV achieves a 29% CoV using 25 phases. At the same number of phases, the BBV+DDV detector reduces CoV to about 15%. The savings on the number of unique phases (and hence tunings) required to achieve a given CoV value are also dramatic: For instance, at a CoV value of 29%, the addition of the DDV reduces the number of phases from 25 to 11. These same trends repeat for 8 processors, where the addition of the DDV consistently improves the CoV values and reduces the tuning overhead by at least half.

Overall, considering data distribution and access patterns across the system leads to better phase detection, improving CoV by factors of two to three in many of configurations studied (for a fixed number of phases), and often reducing tuning overhead by half (for a fixed CoV). The importance of tracking data accesses increases with processor count, which amplifies the benefits of the DDV as systems are scaled up.

V. Related Work

Efforts to better understand and exploit the periodic behavior of programs extend back to Denning's working set studies [2]. More recent work focuses on understanding and predicting the large-scale behavior of applications [7], [5]. Dhodapkar and Smith explore compressed signatures for uniprocessor instruction working sets [3], and compare three online approaches to driving adaptations [4]. They find the BBV signatures of Sherwood et al. [8] to yield the most stable phases and prove most efficient at detecting performance changes. In contrast, instruction working set signatures give longer phases (thus requiring fewer retuning intervals) than either BBVs or a conditional branch count technique introduced by Balasubramonian et al. [1].

VI. Conclusions

We have explored dynamic phase detection in DSM multiprocessors. We have proposed and evaluated a novel hardware extension that builds on the BBV phase detector mechanism originally developed for uniprocessors [8], but yields significant improvements in both the quality and the number of the identified phases over the BBV hardware alone. It does so by tracking the access frequency, contention, and distance to data touched by each processor.

We believe that future work in this direction should move toward combining the insights derived from our study with appropriate phase prediction mechanisms, to ultimately steer hardware reconfiguration of DSM multiprocessors.

References

[1] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *IEEE/ACM 33nd International Symposium on Microarchitecture*, pages 245–257, Dec. 2000.

[2] P. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968.

[3] A. Dhodapkar and J. Smith. Managing multi-configurable hardware via working set analysis. In *29th Annual International Symposium on Computer Architecture*, pages 233–246, May 2002.

[4] A. Dhodapkar and J. Smith. Comparing program phase detection techniques. In *IEEE/ACM 37th Annual International Symposium on Microarchitecture*, pages 217–227, Dec. 2003.

[5] E. Duesterwald, C. Caşcaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 220–231, Sept. 2003.

[6] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *11th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 165–176, Oct. 2004.

[7] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *10th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, Oct. 2002.

[8] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *30th Annual International Symposium on Computer Architecture*, pages 336–349, June 2003.

[9] Standard Performance Evaluation Corporation. SPEC OMP benchmark suite. `http://www.specbench.org/hpg/omp2001/`, 2001.

[10] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.

This work was performed under the auspices of the U. S. Department of Energy by University of California, Lawrence Livermore National Laboratory under contract W-7405-Eng-48.